Week 12.3

2303A51018

B- 28

Task 1: Sorting Student Records for Placement Drive

Scenario

SR University's Training and Placement Cell needs to shortlist candidates efficiently during campus placements. Student records must be sorted by CGPA in descending order.

Tasks

1. Use GitHub Copilot to generate a program that stores student records (Name, Roll Number, CGPA).

2. Implement the following sorting algorithms using AI assistance:

o Quick Sort

o Merge Sort

3. Measure and compare runtime performance for large datasets.

4. Write a function to display the top 10 students based on CGPA.

Expected Outcome

• Correctly sorted student records.

• Performance comparison between Quick Sort and Merge Sort.

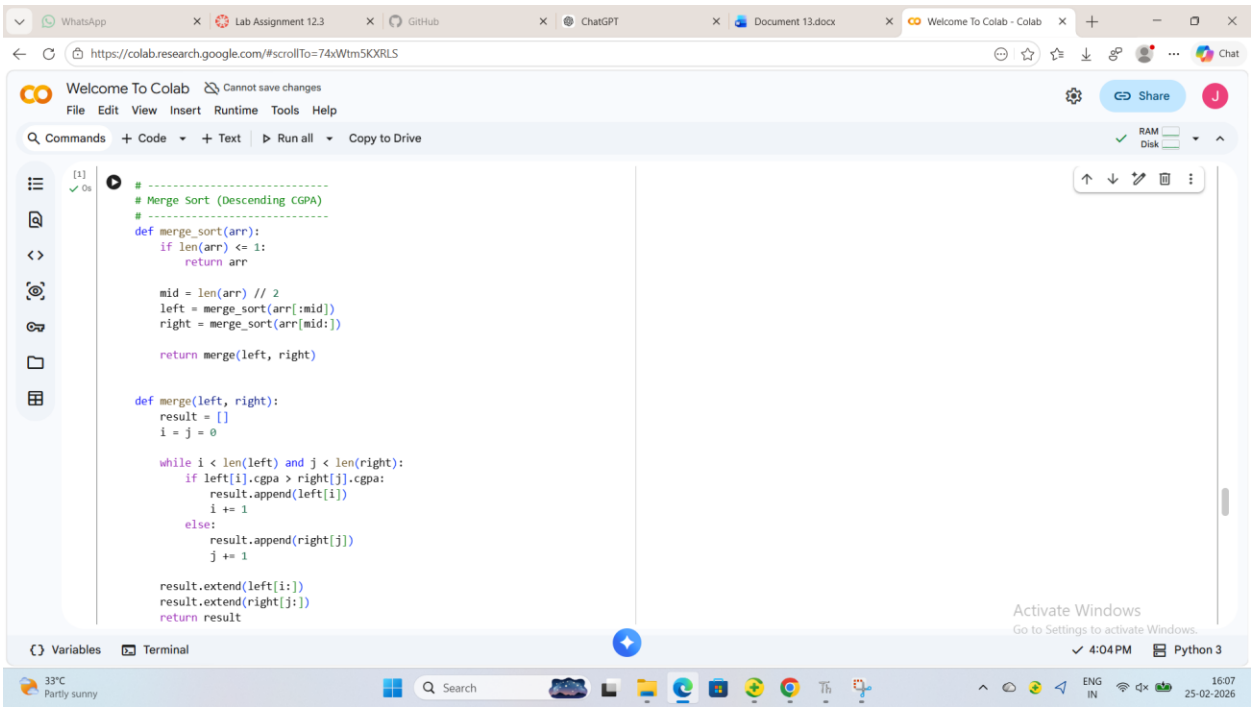
• Clear output of top-performing students.

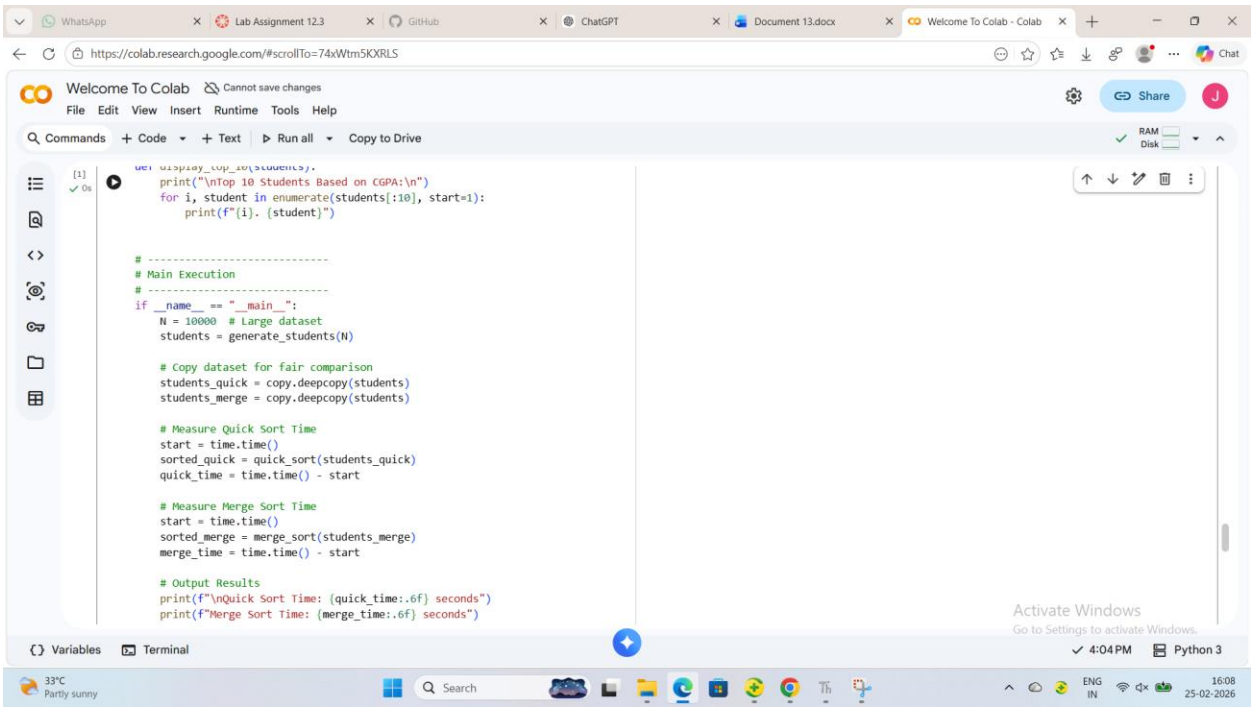
Code

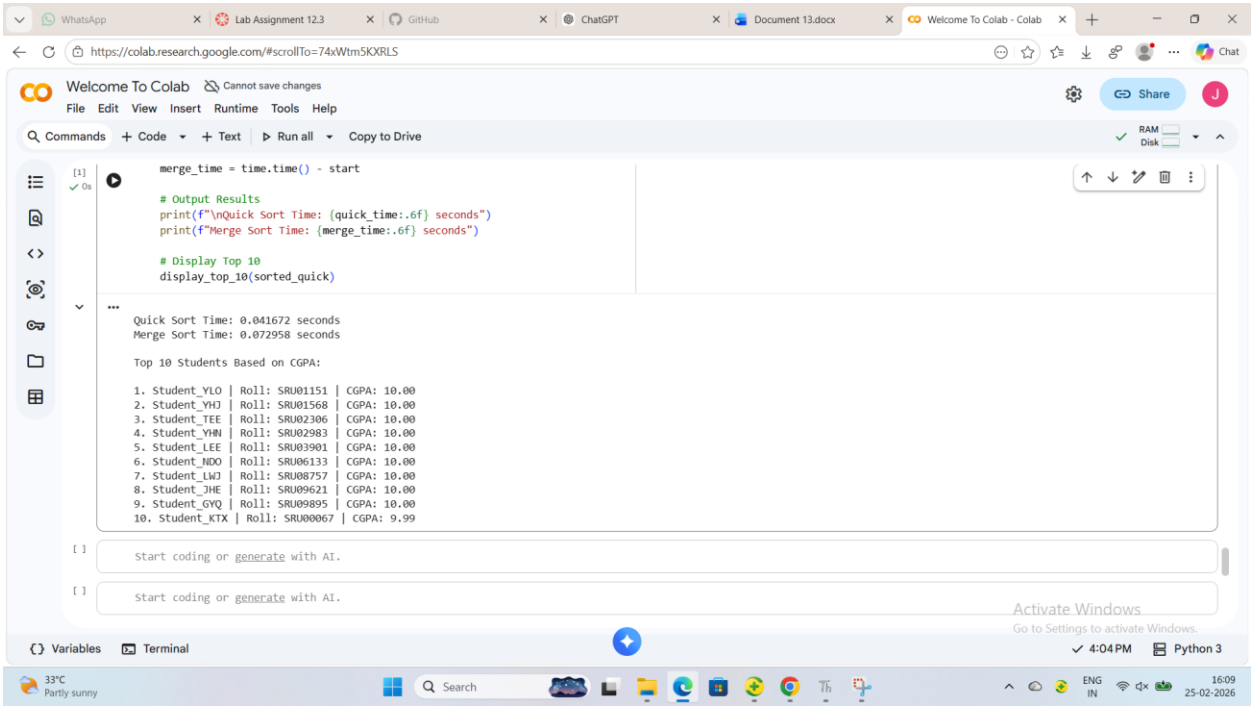
[1]
✓ 0s

```python
import random
import string
import time
import copy

# ------------------------------
# Student Record Structure
# ------------------------------
class Student:
    def __init__(self, name, roll, cgpa):
        self.name = name
        self.roll = roll
        self.cgpa = cgpa

    def __repr__(self):
        return f"{self.name} | Roll: {self.roll} | CGPA: {self.cgpa:.2f}"


# ------------------------------
# Generate Large Dataset
# ------------------------------
def generate_students(n):
    students = []
    for i in range(n):
        name = "Student_" + ''.join(random.choices(string.ascii_uppercase, k=3))
        roll = f"SRU{i+1:05d}"
        cgpa = round(random.uniform(5.0, 10.0), 2)
        students.append(Student(name, roll, cgpa))
```

```
 return students


----------------------------
uick Sort (Descending CGPA)
----------------------------
 quick_sort(arr):
 if len(arr) <= 1:
     return arr

 pivot = arr[len(arr)//2].cgpa
 left = [x for x in arr if x.cgpa > pivot]
 middle = [x for x in arr if x.cgpa == pivot]
 right = [x for x in arr if x.cgpa < pivot]

 return quick_sort(left) + middle + quick_sort(rig


----------------------------
erge Sort (Descending CGPA)
----------------------------
 merge_sort(arr):
 if len(arr) <= 1:
     return arr

 mid = len(arr) // 2
 left = merge_sort(arr[:mid])
 right = merge_sort(arr[mid:])
```


Welcome To Colab
Cannot save changes
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text Run all Copy to Drive
RAM
Disk
[1]
0s
# ----------------------------
# Merge Sort (Descending CGPA)
# ----------------------------
def merge_sort(arr):
if len(arr) <= 1:
return arr
mid = len(arr) // 2
left = merge_sort(arr[:mid])
right = merge_sort(arr[mid:])
return merge(left, right)
def merge(left, right):
result = []
i = j = 0
while i < len(left) and j < len(right):
if left[i].cgpa > right[j].cgpa:
result.append(left[i])
i += 1
else:
result.append(right[j])
j += 1
result.extend(left[i:])
result.extend(right[j:])
return result
Activate Windows
Go to Settings to activate Windows.
Variables
Terminal
4:04 PM
Python 3
33°C
Partly sunny
Search
ENG
IN
16:07
25-02-2026





Explanation

The goal is to:

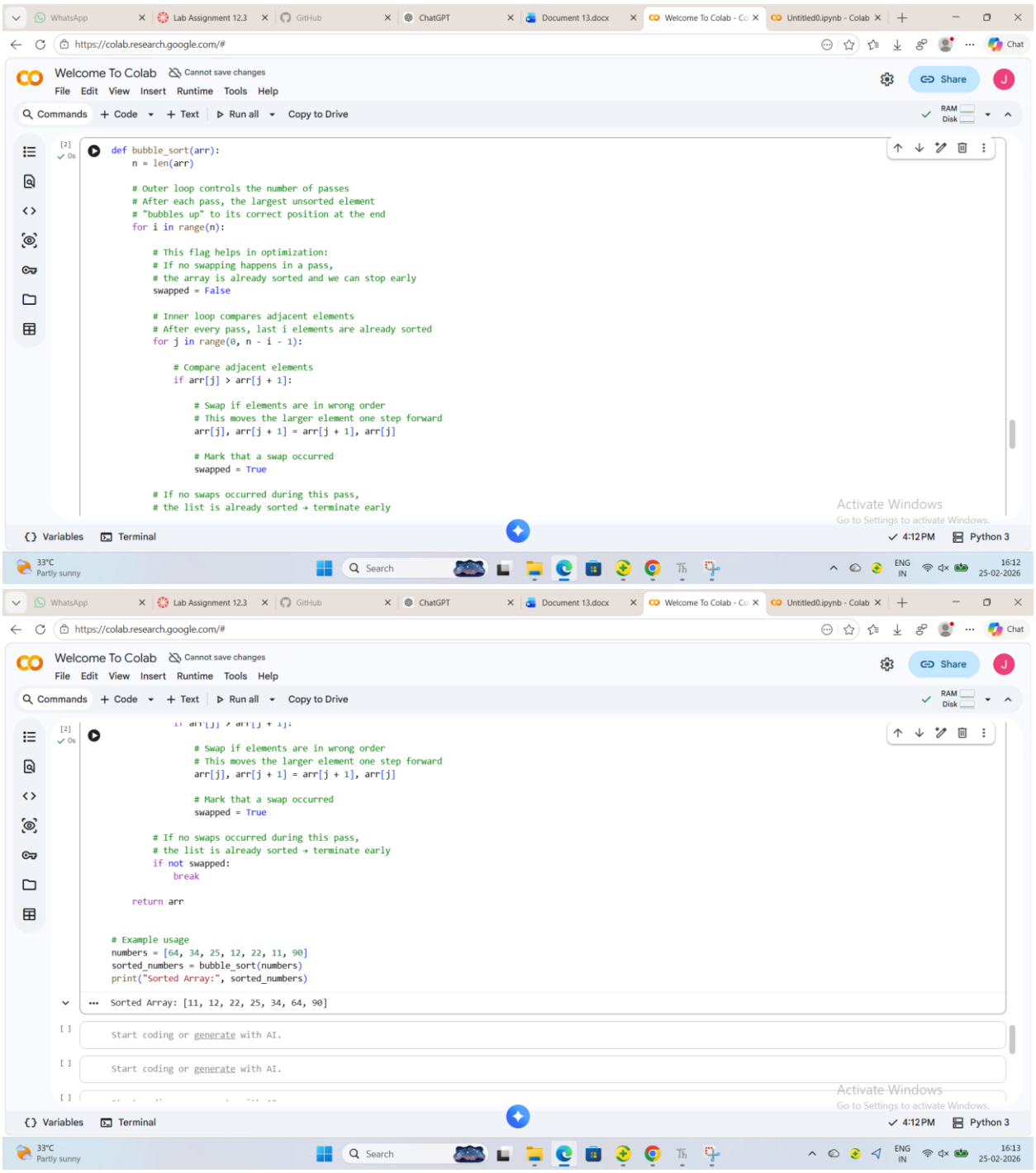
- Store student records (Name, Roll Number, CGPA)
- Sort students by **CGPA in descending order**
- Compare **Quick Sort** and **Merge Sort** performance

- Display the **Top 10 students**
-

Task 2: Implementing Bubble Sort with AI Comments

- **Task: Write a Python implementation of Bubble Sort.**
- **Instructions:**
- **Students implement Bubble Sort normally.**
- **Ask AI to generate inline comments explaining key logic (like swapping, passes, and termination).**
- **Request AI to provide time complexity analysis.**
- **Expected Output:**
- **A Bubble Sort implementation with AI-generated explanatory comments and complexity analysis.**

Code



Explanation

The goal of this task is to:

- Implement the **Bubble Sort algorithm**
- Add AI-generated inline comments explaining:
    - Swapping logic

    - Passes
    - Termination condition
- Provide time complexity analysis

What is Bubble Sort?

Bubble Sort is a **simple comparison-based sorting algorithm**.

It works by:

- Repeatedly comparing **adjacent elements**
- Swapping them if they are in the wrong order
- Continuing this process until the list is sorted

It is called *Bubble Sort* because the largest elements "bubble up" to the end of the list after each pass.

2 How the Algorithm Works (Step-by-Step)

Assume the list:

```
[5, 3, 8, 4]
```

◇ Pass 1:

- Compare 5 and 3 → Swap → [3, 5, 8, 4]
- Compare 5 and 8 → No swap
- Compare 8 and 4 → Swap → [3, 5, 4, 8]

Now, **8 is in correct position**.

◇ Pass 2:

- Compare 3 and 5 → No swap

- Compare 5 and 4 → Swap → [3, 4, 5, 8]

Now, **5 is in correct position**.

◇ Pass 3:

- Compare 3 and 4 → No swap

No swaps happened → Stop early.

Task 3: Quick Sort and Merge Sort Comparison

• Task: Implement Quick Sort and Merge Sort using recursion.

• Instructions:

• Provide AI with partially completed functions for recursion.

• Ask AI to complete the missing logic and add docstrings.

• Compare both algorithms on random, sorted, and reverse-sorted

lists.

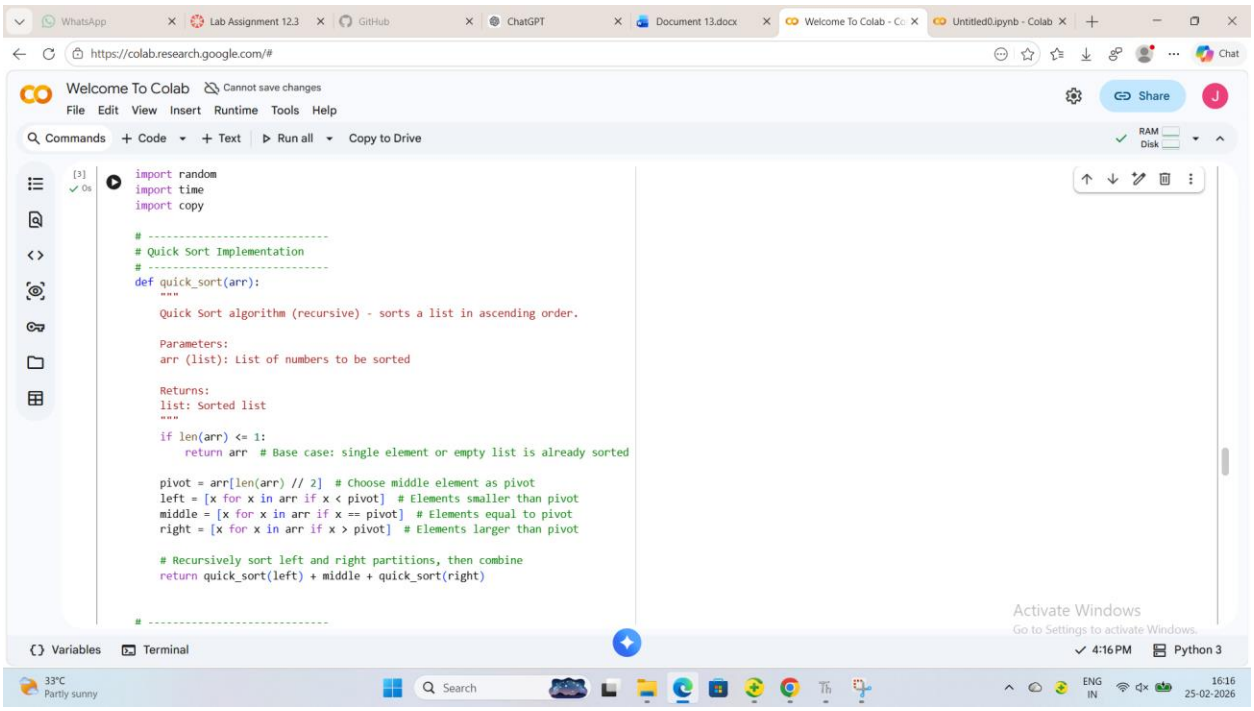
• Expected Output:

• Working Quick Sort and Merge Sort implementations.

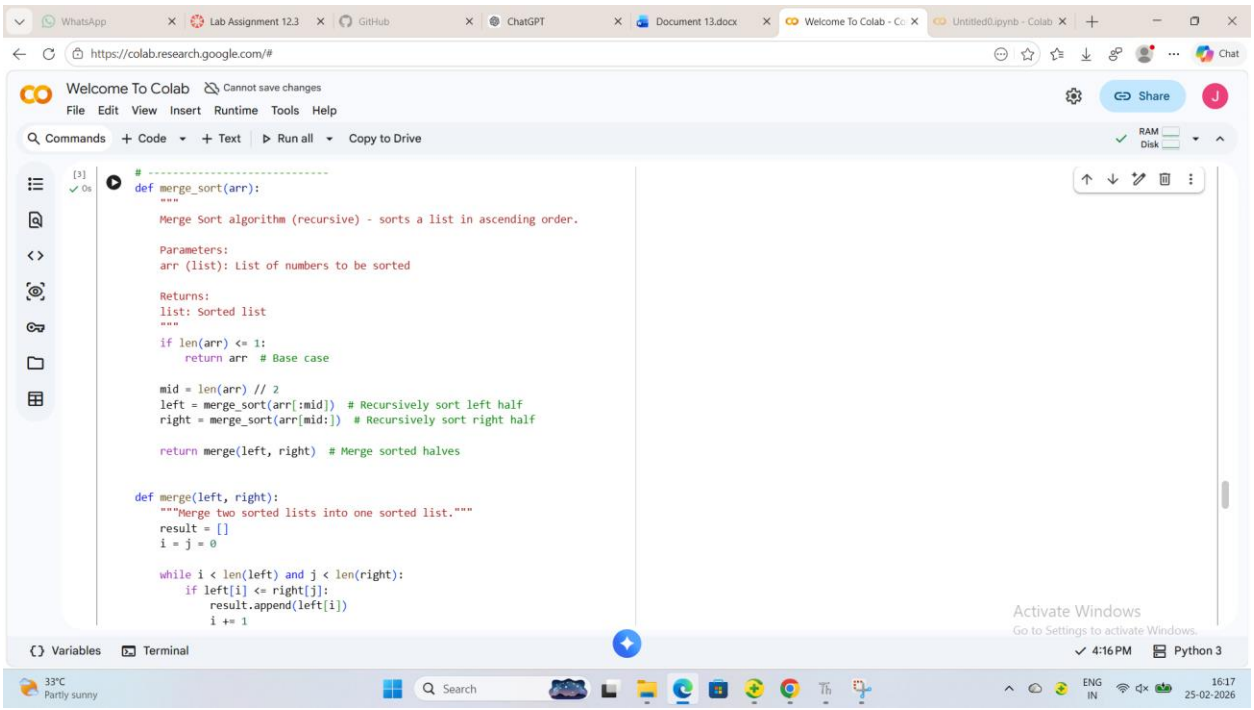
• AI-generated explanation of average, best, and worst-case

Complexities.

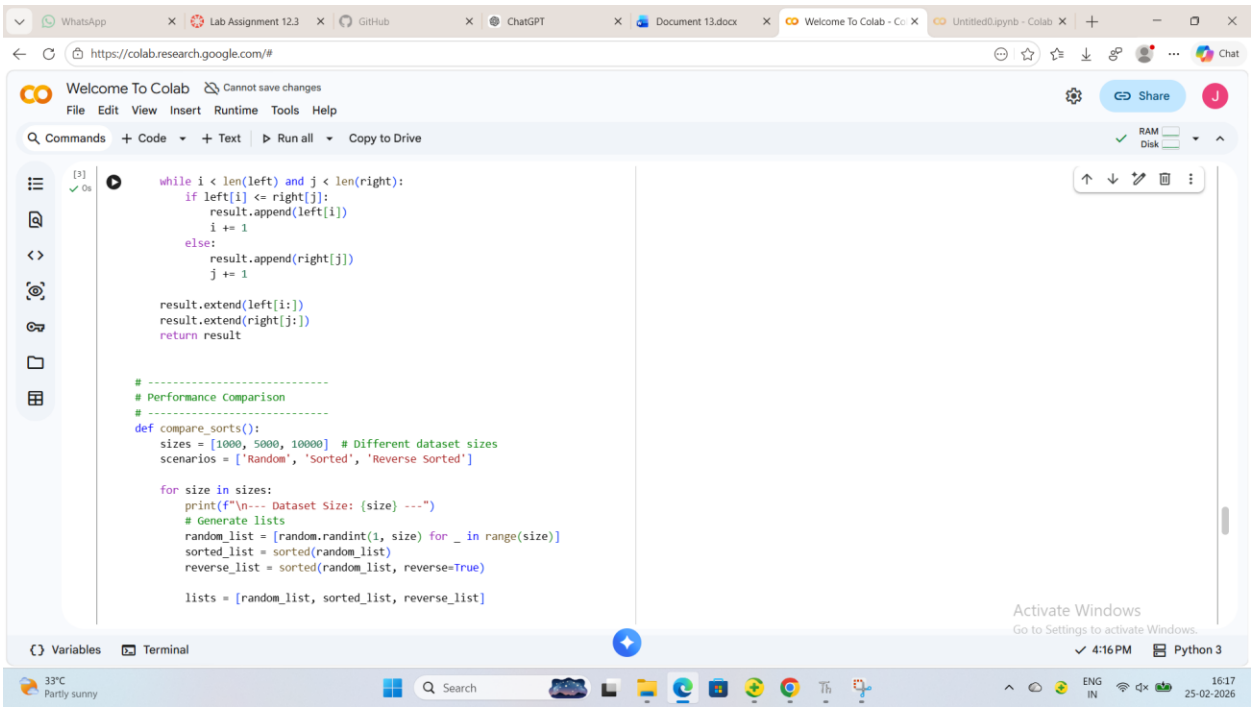
Code



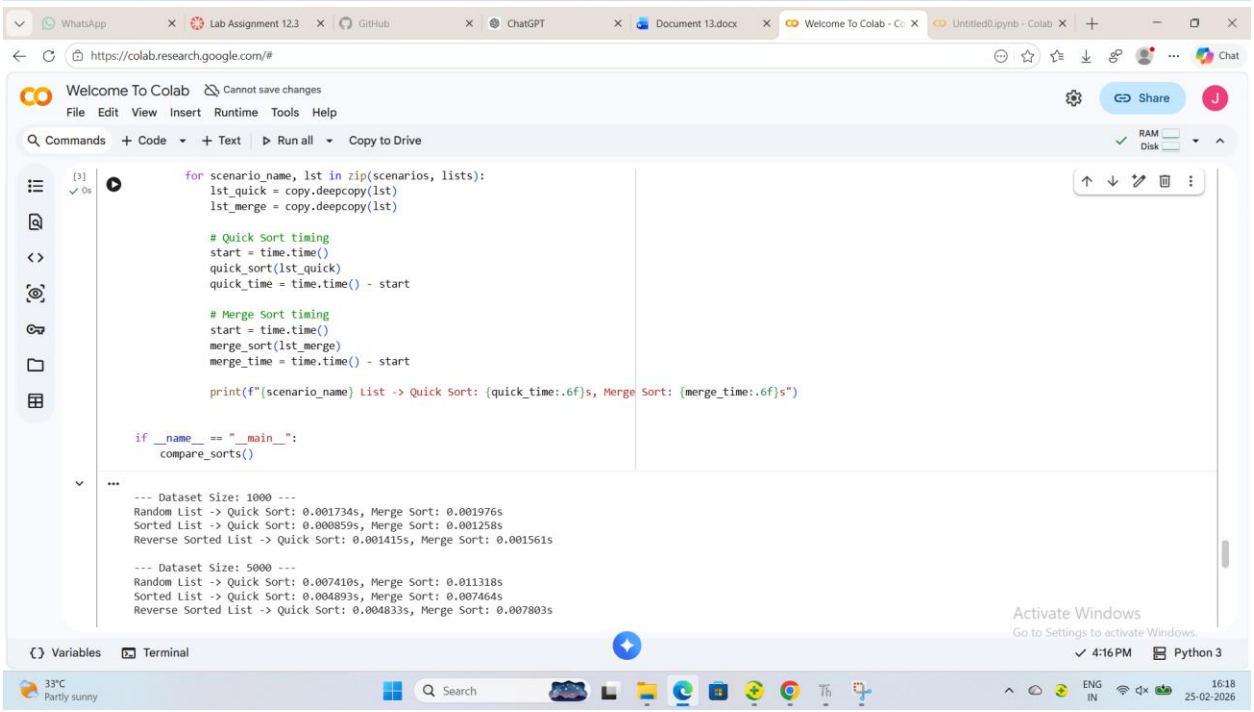
```
import random
import time
import copy

# ----------------------------
# Quick Sort Implementation
# ----------------------------
def quick_sort(arr):
    """
    Quick Sort algorithm (recursive) - sorts a list in ascending order.

    Parameters:
    arr (list): List of numbers to be sorted

    Returns:
    list: Sorted list
    """
    if len(arr) <= 1:
        return arr  # Base case: single element or empty list is already sorted

    pivot = arr[len(arr) // 2]  # Choose middle element as pivot
    left = [x for x in arr if x < pivot]  # Elements smaller than pivot
    middle = [x for x in arr if x == pivot]  # Elements equal to pivot
    right = [x for x in arr if x > pivot]  # Elements larger than pivot

    # Recursively sort left and right partitions, then combine
    return quick_sort(left) + middle + quick_sort(right)


# ----------------------------
```



```
# ----------------------------
def merge_sort(arr):
    """
    Merge Sort algorithm (recursive) - sorts a list in ascending order.

    Parameters:
    arr (list): List of numbers to be sorted

    Returns:
    list: Sorted list
    """
    if len(arr) <= 1:
        return arr  # Base case

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])  # Recursively sort left half
    right = merge_sort(arr[mid:])  # Recursively sort right half

    return merge(left, right)  # Merge sorted halves


def merge(left, right):
    """Merge two sorted lists into one sorted list."""
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
```

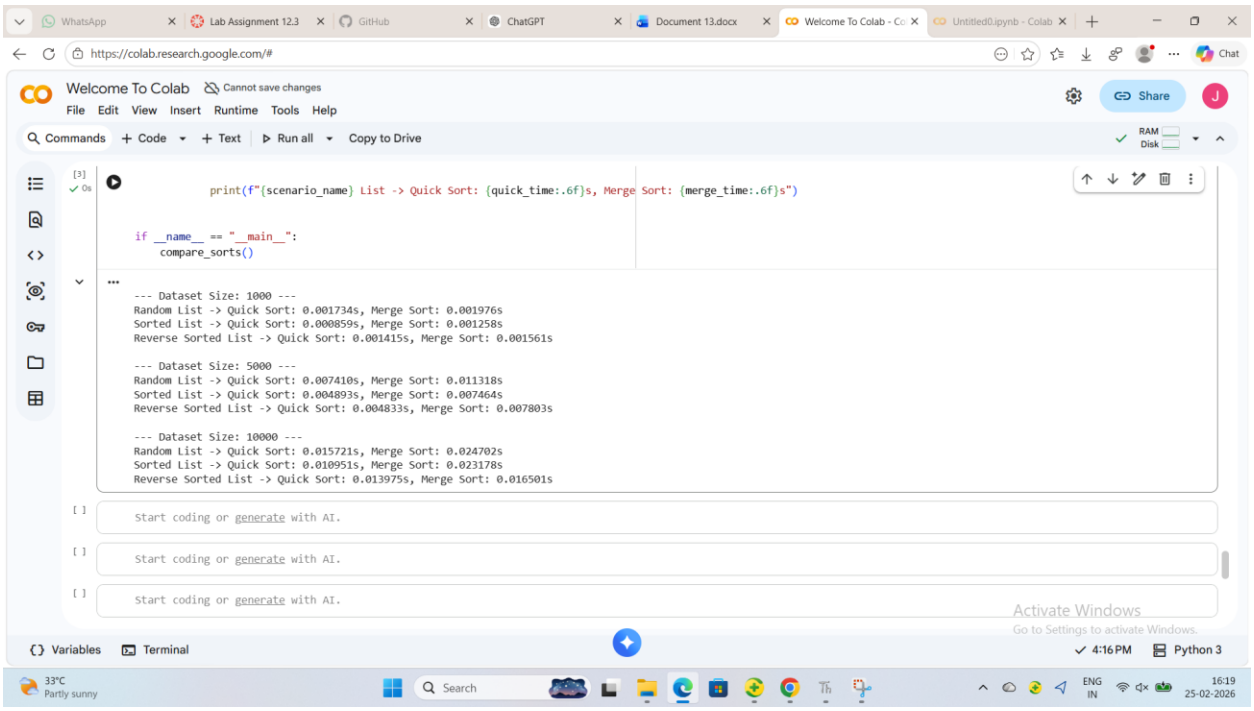


```
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result


# ----------------------------
# Performance Comparison
# ----------------------------
def compare_sorts():
    sizes = [1000, 5000, 10000]  # Different dataset sizes
    scenarios = ['Random', 'Sorted', 'Reverse Sorted']

    for size in sizes:
        print(f"\n--- Dataset Size: {size} ---")
        # Generate lists
        random_list = [random.randint(1, size) for _ in range(size)]
        sorted_list = sorted(random_list)
        reverse_list = sorted(random_list, reverse=True)

        lists = [random_list, sorted_list, reverse_list]
```



```
        for scenario_name, lst in zip(scenarios, lists):
            lst_quick = copy.deepcopy(lst)
            lst_merge = copy.deepcopy(lst)

            # Quick Sort timing
            start = time.time()
            quick_sort(lst_quick)
            quick_time = time.time() - start

            # Merge Sort timing
            start = time.time()
            merge_sort(lst_merge)
            merge_time = time.time() - start

            print(f"{scenario_name} List -> Quick Sort: {quick_time:.6f}s, Merge Sort: {merge_time:.6f}s")


if __name__ == "__main__":
    compare_sorts()
```

```
--- Dataset Size: 1000 ---
Random List -> Quick Sort: 0.001734s, Merge Sort: 0.001976s
Sorted List -> Quick Sort: 0.000859s, Merge Sort: 0.001258s
Reverse Sorted List -> Quick Sort: 0.001415s, Merge Sort: 0.001561s

--- Dataset Size: 5000 ---
Random List -> Quick Sort: 0.007410s, Merge Sort: 0.011318s
Sorted List -> Quick Sort: 0.004893s, Merge Sort: 0.007464s
Reverse Sorted List -> Quick Sort: 0.004833s, Merge Sort: 0.007803s
```



Explanation

Quick Sort:

- **Algorithm Type:** Divide and conquer, recursive
- **Logic:** Pick a pivot → Partition array → Recursively sort left & right → Combine
- **Average Case:** O(n log n)
- **Best Case:** O(n log n)
- **Worst Case:** $O(n^2)$ (e.g., already sorted list with poor pivot selection)
- **Space Complexity:** O(log n) recursion stack

Merge Sort:

- **Algorithm Type:** Divide and conquer, recursive, stable
- **Logic:** Divide list in halves → Recursively sort → Merge halves
- **Average/Best/Worst Case:** O(n log n) consistently
- **Space Complexity:** O(n) extra memory for merging

Task 4 (Real-Time Application – Inventory Management System)

Scenario: A retail store's inventory system contains thousands of products,

each with attributes like product ID, name, price, and stock quantity. Store staff

need to:

1. Quickly search for a product by ID or name.

2. Sort products by price or quantity for stock analysis.

Task:

• Use AI to suggest the most efficient search and sort algorithms for this use case.

• Implement the recommended algorithms in Python.

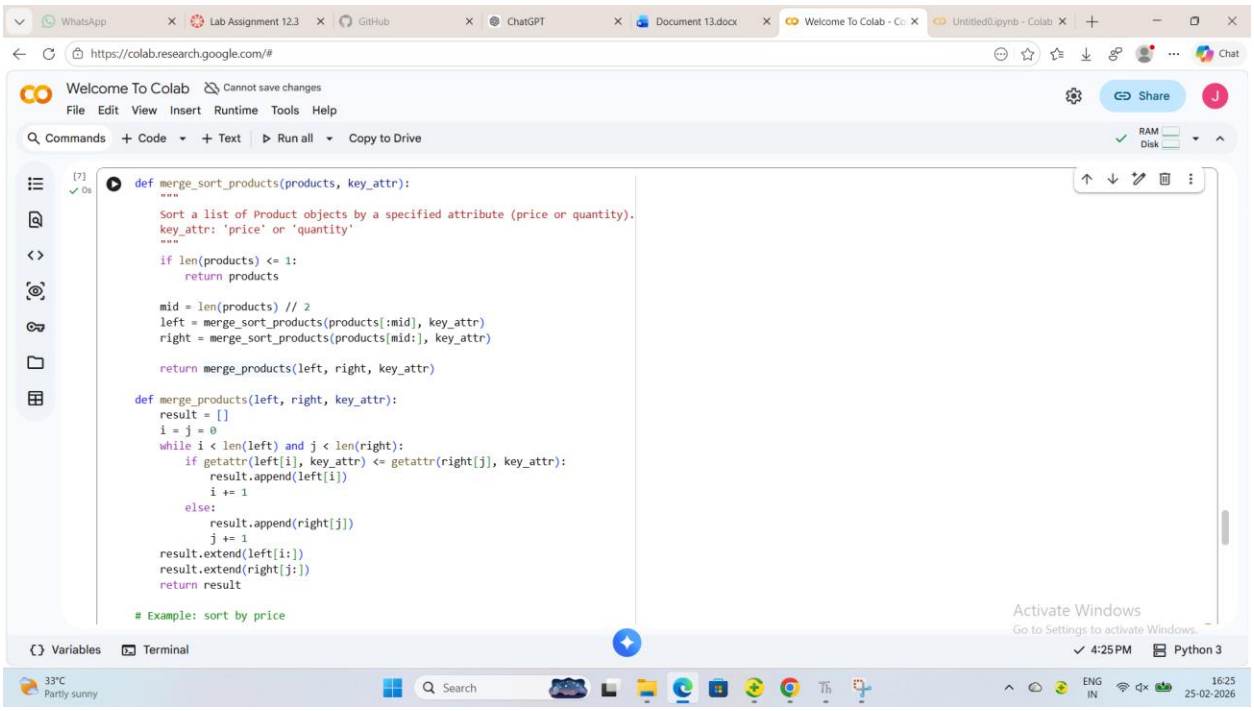
• Justify the choice based on dataset size, update frequency, and performance requirements.

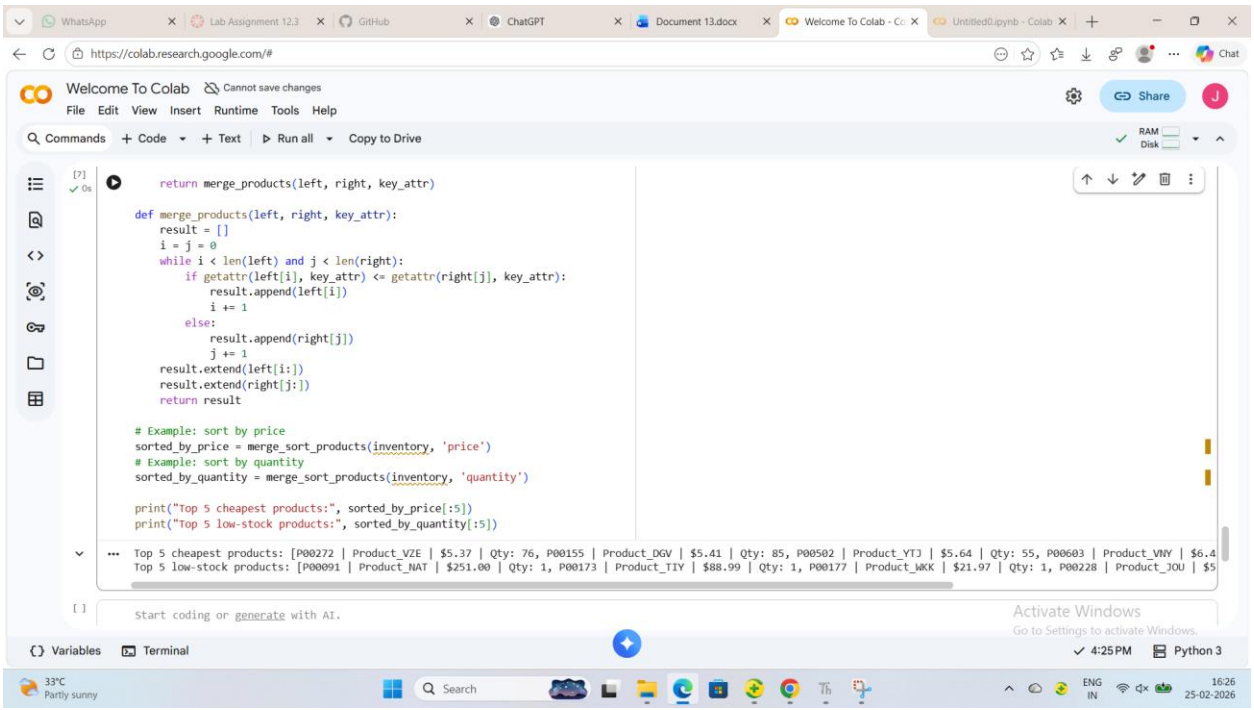
Expected Output:

• A table mapping operation → recommended algorithm → justification.

• Working Python functions for searching and sorting the inventory.

Code





Justification of Algorithm Choices

1. **Search by ID → Dictionary**
    a. Product IDs are unique → O(1) average lookup
    b. Ideal for frequent lookups in real-time operations
2. **Search by Name → Binary Search**

a. Names can be sorted alphabetically
b. Binary search gives O(log n) lookup
c. Requires inventory sorted by name, efficient for thousands of products

3. **Sorting → Merge Sort**
    a. Handles large datasets efficiently (O(n log n))
    b. Stable sort (keeps relative order if multiple products have same price)
    c. Suitable for periodic analysis (price ranking, stock monitoring)

Task 5: Real-Time Stock Data Sorting & Searching

Scenario:

An AI-powered FinTech Lab at SR University is building a tool for analyzing stock price movements. The requirement is to quickly sort stocks by daily gain/loss and search for specific stock symbols efficiently.

• Use GitHub Copilot to fetch or simulate stock price data (Stock Symbol, Opening Price, Closing Price).

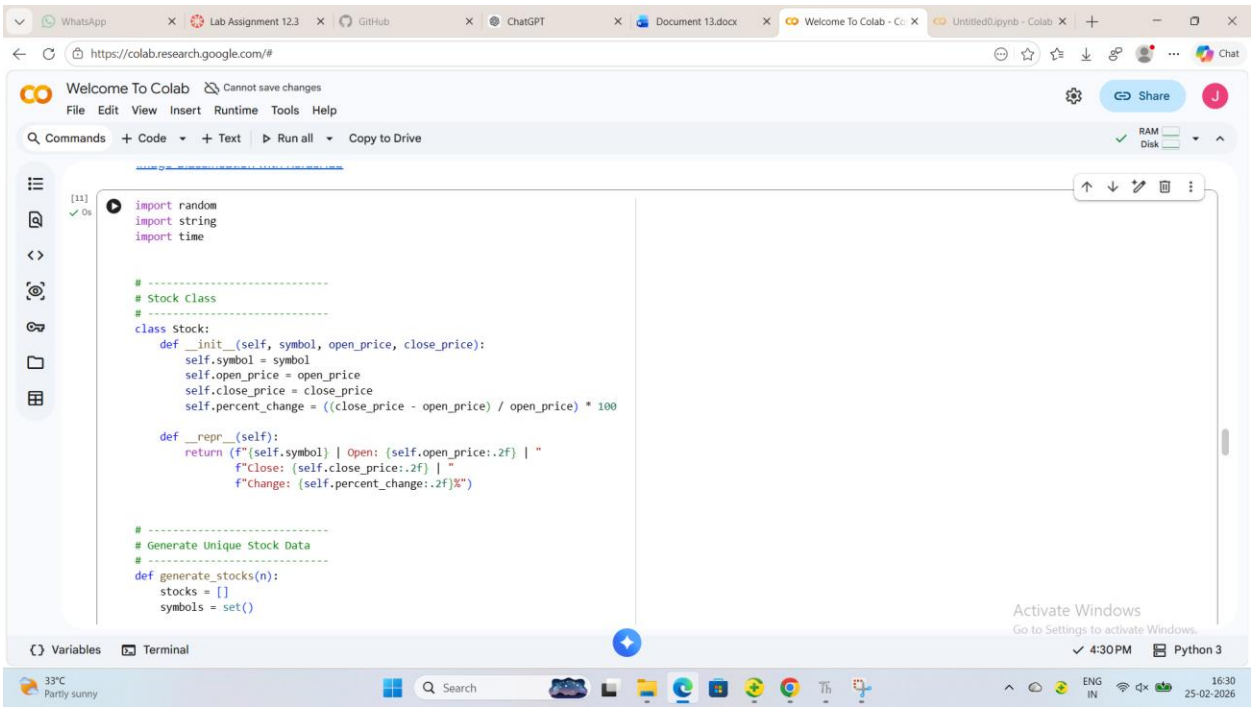
• Implement sorting algorithms to rank stocks by percentage change.

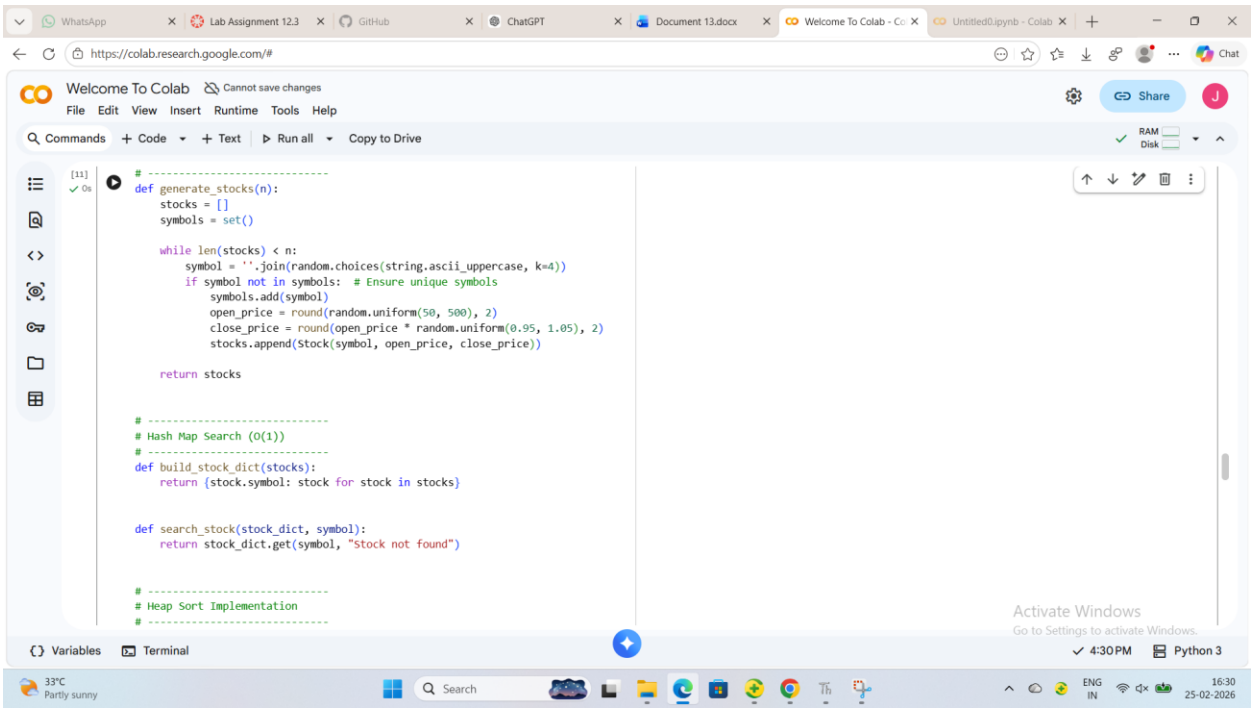
• Implement a search function that retrieves stock data instantly when a stock symbol is entered.

• Optimize sorting with Heap Sort and searching with Hash Maps.

• Compare performance with standard library functions (sorted(), dict lookups) and analyze trade-offs.

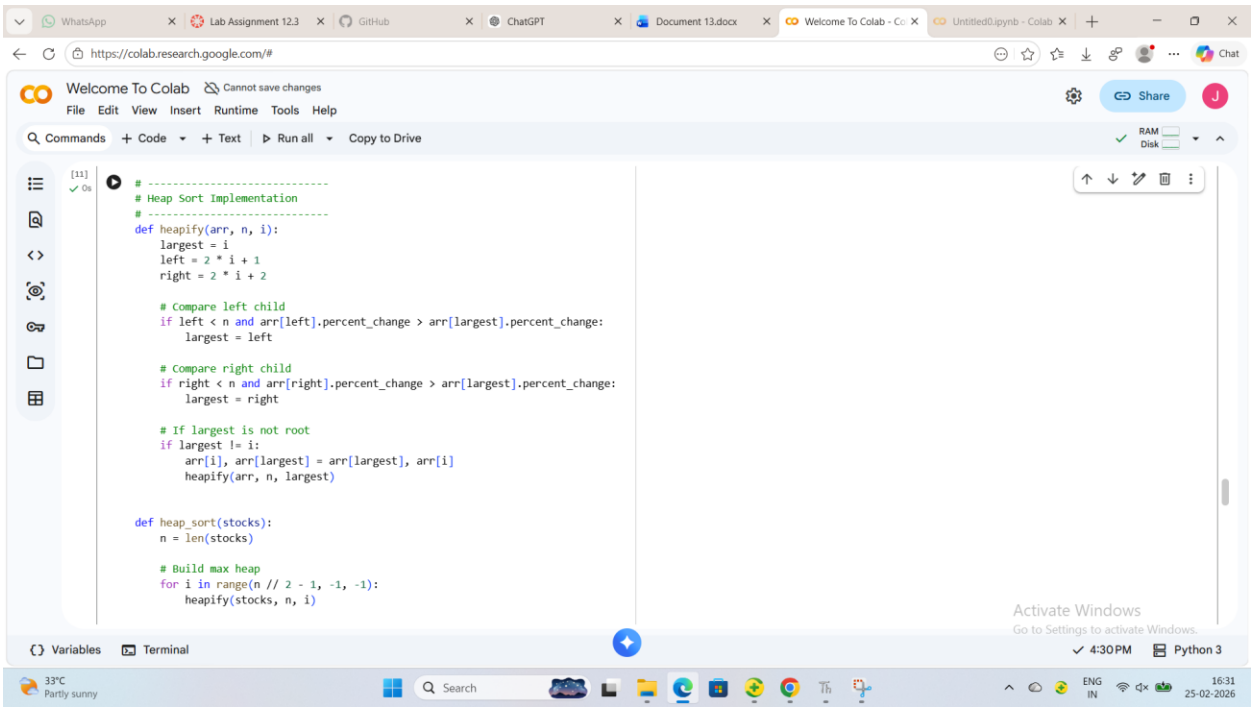
Code

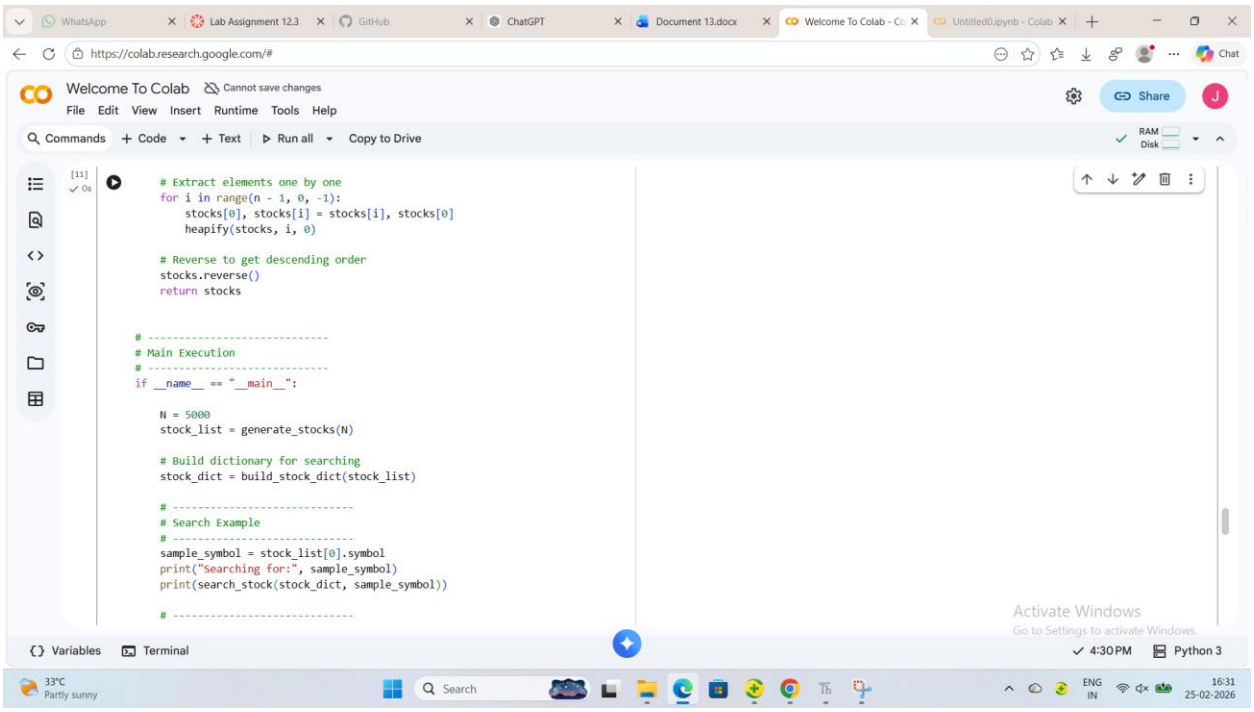

Welcome To Colab
Cannot save changes
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text Run all Copy to Drive
RAM Disk
[11] 0s
import random
import string
import time
# ------------------------------
# Stock Class
# ------------------------------
class Stock:
def __init__(self, symbol, open_price, close_price):
self.symbol = symbol
self.open_price = open_price
self.close_price = close_price
self.percent_change = ((close_price - open_price) / open_price) * 100
def __repr__(self):
return (f"{self.symbol} | Open: {self.open_price:.2f} | "
f"Close: {self.close_price:.2f} | "
f"Change: {self.percent_change:.2f}%")
# ------------------------------
# Generate Unique Stock Data
# ------------------------------
def generate_stocks(n):
stocks = []
symbols = set()
Activate Windows
Go to Settings to activate Windows.
Variables Terminal
4:30 PM Python 3

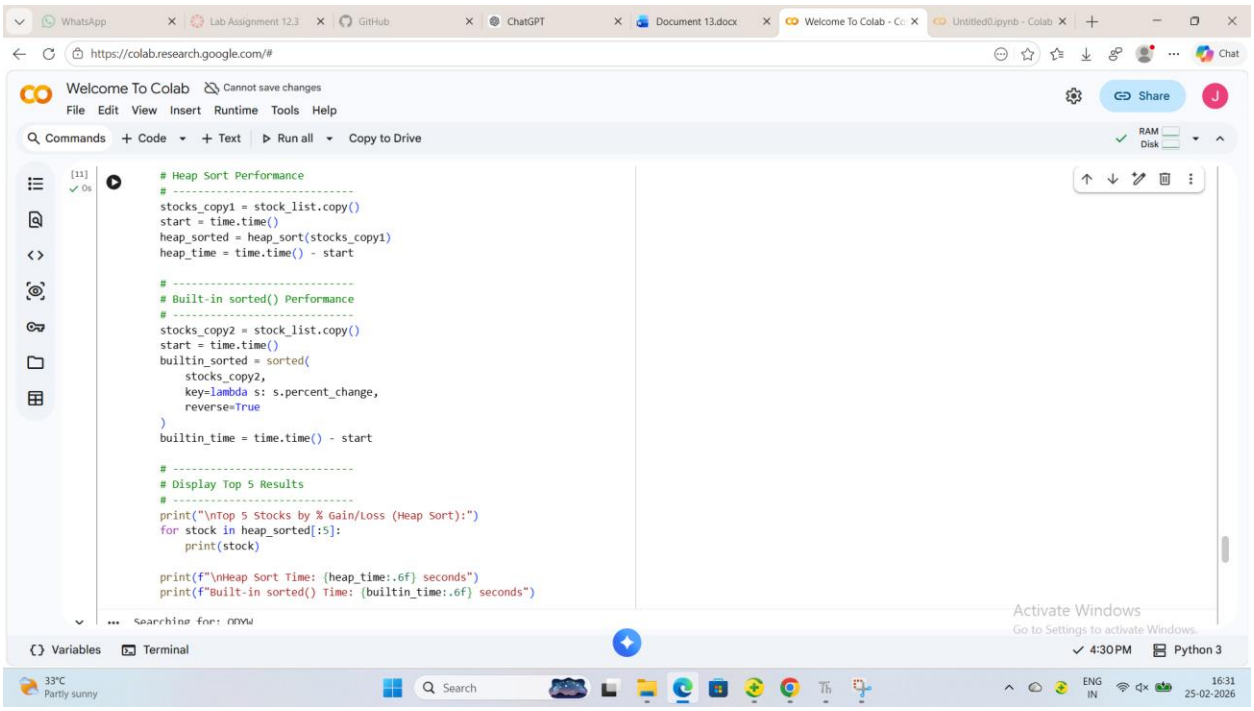

Welcome To Colab
Cannot save changes
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text Run all Copy to Drive
RAM Disk
[11] 0s
# ------------------------------
def generate_stocks(n):
stocks = []
symbols = set()
while len(stocks) < n:
symbol = ''.join(random.choices(string.ascii_uppercase, k=4))
if symbol not in symbols: # Ensure unique symbols
symbols.add(symbol)
open_price = round(random.uniform(50, 500), 2)
close_price = round(open_price * random.uniform(0.95, 1.05), 2)
stocks.append(Stock(symbol, open_price, close_price))
return stocks
# ------------------------------
# Hash Map Search (O(1))
# ------------------------------
def build_stock_dict(stocks):
return {stock.symbol: stock for stock in stocks}
def search_stock(stock_dict, symbol):
return stock_dict.get(symbol, "Stock not found")

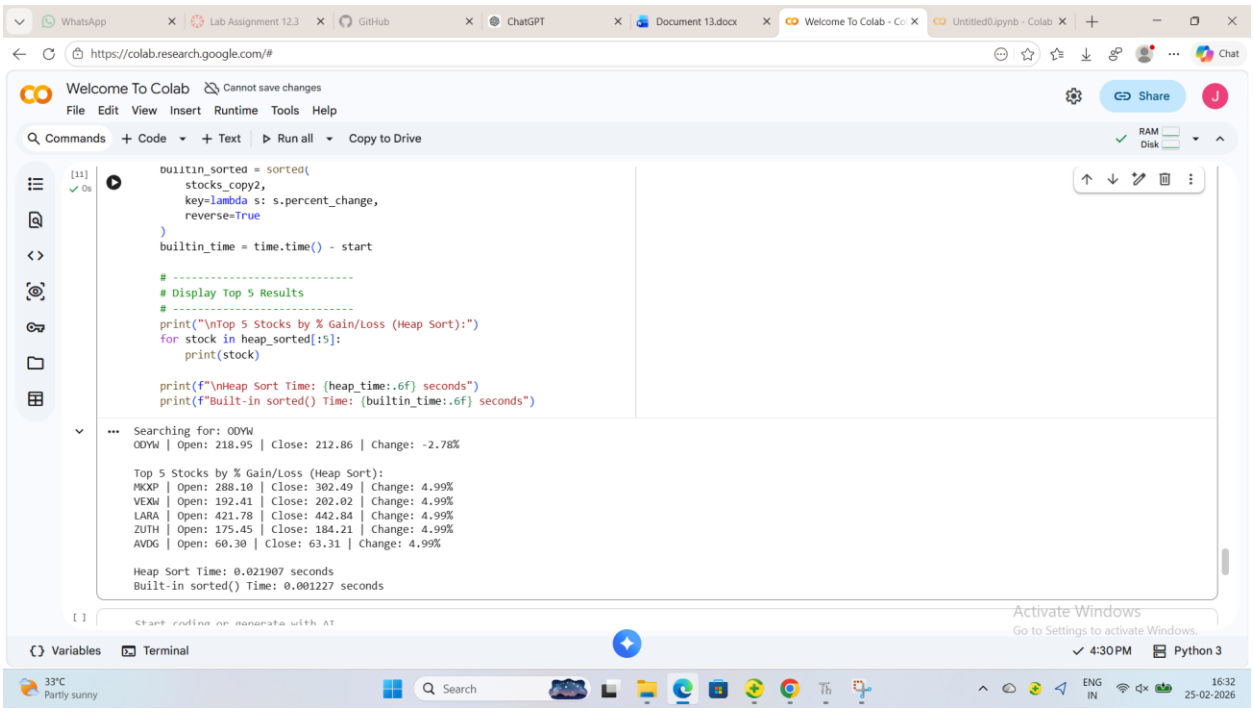
Heap Sort Implementation

Activate Windows
Go to Settings to activate Windows.
Variables Terminal
4:30 PM Python 3


Welcome To Colab
Cannot save changes
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text Run all Copy to Drive
RAM
Disk
[11]
0s
# ----------------------------
# Heap Sort Implementation
# ----------------------------
def heapify(arr, n, i):
largest = i
left = 2 * i + 1
right = 2 * i + 2
# Compare left child
if left < n and arr[left].percent_change > arr[largest].percent_change:
largest = left
# Compare right child
if right < n and arr[right].percent_change > arr[largest].percent_change:
largest = right
# If largest is not root
if largest != i:
arr[i], arr[largest] = arr[largest], arr[i]
heapify(arr, n, largest)
def heap_sort(stocks):
n = len(stocks)
# Build max heap
for i in range(n // 2 - 1, -1, -1):
heapify(stocks, n, i)
Activate Windows
Go to Settings to activate Windows.
Variables Terminal
4:30 PM Python 3
33°C
Partly sunny
Search
ENG
IN
16:31
25-02-2026


Welcome To Colab
Cannot save changes
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text Run all Copy to Drive
RAM
Disk
[11]
0s
# Extract elements one by one
for i in range(n - 1, 0, -1):
stocks[0], stocks[i] = stocks[i], stocks[0]
heapify(stocks, i, 0)
# Reverse to get descending order
stocks.reverse()
return stocks
# ----------------------------
# Main Execution
# ----------------------------
if __name__ == "__main__":
N = 5000
stock_list = generate_stocks(N)
# Build dictionary for searching
stock_dict = build_stock_dict(stock_list)
# ----------------------------
# Search Example
# ----------------------------
sample_symbol = stock_list[0].symbol
print("Searching for:", sample_symbol)
print(search_stock(stock_dict, sample_symbol))
# ----------------------------
Activate Windows
Go to Settings to activate Windows.
Variables Terminal
4:30 PM Python 3
33°C
Partly sunny
Search
ENG
IN
16:31
25-02-2026





Scenario Summary

- Dataset: Stocks with
  - Stock Symbol
  - Opening Price

    - Closing Price
- Requirements:
    - Sort stocks by **daily percentage change** (gain/loss)
    - **Instant search** by stock symbol
    - Optimize performance using **Heap Sort** (sorting) and **Hash Maps** (searching)

The goal is to:

1. Simulate stock price data (Symbol, Opening Price, Closing Price)
2. Calculate daily percentage change
3. Sort stocks by gain/loss efficiently
4. Search stock symbols instantly
5. Compare custom algorithms with Python built-in functions

Stock Data Simulation

We create a `Stock` class containing:

- `symbol`
- `open_price`
- `close_price`
- `percent_change`

Percentage Change Formula:

$Percent\ Change = \frac{(Close - Open)}{Open} \times 100$Percent Change=Open(Close−Open) ×100

This value helps rank stocks by daily gain or loss.

2 Searching Optimization (Hash Map)

◇ Why Use a Dictionary?

We store stocks in a dictionary:

```
stock_dict = {stock.symbol: stock}
```

This allows:

- **O(1) average time complexity**
- Instant lookup
- Efficient for real-time systems

◇ Example

If user enters:

```
search_stock("AAPL")
```

The result is retrieved immediately without scanning the entire list.

3 Sorting Optimization (Heap Sort)

◇ Why Heap Sort?

Heap Sort:

- Time Complexity: **O(n log n)**
- Good for large datasets
- Useful when repeatedly finding top gainers/losers

◇ How It Works:

1. Build a **Max Heap** using percentage change.
2. Extract largest element repeatedly.
3. Reverse list for descending order ranking.

This ranks stocks from highest gain to lowest.

4 Performance Comparison

We compare:

Method	Description
Heap Sort	Manual implementation
sorted()	Python built-in (Timsort)

Observations:

- `sorted()` is usually slightly faster because it is highly optimized in C.
- Heap Sort is useful when:
    - Frequently extracting top N elements
    - Working with streaming data

5 Time Complexity Analysis

◇ Searching

- Hash Map Lookup → **O(1)** average
- Very efficient for real-time queries

◇ Sorting

Algorithm	Best	Average	Worst
Heap Sort	O(n log n)	O(n log n)	O(n log n)
sorted() (Timsort)	O(n)	O(n log n)	O(n log n)

6 Trade-offs

Feature	Heap Sort	sorted()
Speed	Slightly slower	Faster

Memory	In-place	Uses extra memory
Streaming Top-N	Efficient	Needs full re-sort
Practical Use	Good for repeated ranking	Best for general sorting

7️⃣ Real-World Justification (FinTech Context)

In an AI-powered stock analysis system:

- 🔹 Traders need **instant search** → Hash Map
- 🔹 Analysts need top gainers quickly → Heap structure
- 🔹 Full ranking for reports → sorted()

This makes the system:

- Scalable
- Efficient
- Suitable for thousands of stocks